

# Grundaufbau

LMF2Root kann Cobold LMF-Dateien einlesen und root-Files schreiben. Es können max. 3 Detektoren benutzt werden, nämlich Projektil-, Recoil- und Elektronendetektor. Damit kann man in der Regel Messungen von Ion/Atom-Stoß-, Synchrotron- oder Laser-COLTRIMS auswerten. LMF2Root wird über ein config-File gesteuert. Per Default sucht das Programm nach dem file „config.txt“ in dem Verzeichnis, in dem es gestartet wurde. Man kann aber auch in der Kommandozeile ein anderes config-File angeben, wenn dieses statt config.txt benutzt werden soll. Einfach *lmf2root name.txt* ausführen.

Das config-File ist folgendermaßen aufgebaut: Es gibt am Anfang einer Zeile eine Zahl, das ist die Parameternummer. Die nächste Zahl, die in der Zeile steht ist dann der Wert, den der Parameter haben soll. Alles was nach dieser Zahl kommt wird als Kommentar angesehen. Es gibt max. 10000 Parameter. Im Moment ist die Liste so aufgeteilt, daß Parameter <90 allgemeine Programmeinstellungen sind. Von 100 bis 199 finden sich die Parameter des Projektildetektors. Der Recoildetektor hat die Parameter 200 bis 299, der Elektronendetektor 300 bis 399.

Man kann beliebig eigene Parameter definieren und im Code auf diese zugreifen.

Parameter 10000 ist das Ende der Parameterliste. Danach kommen 3 Blöcke für die Zeitsummenkorrektur der 3 möglichen Detektoren. Jeder diese Blöcke besteht aus den Datenpunkten für die Zeitsummenkorrektur der 3 möglichen Layer und vorangestellt der Information wieviele Punkte vorliegen. Nach diesen 3 Blöcken erwartet das Programm den Filenamen für das Ausgabefile. An den angegebenen Namen wird „root“ angehängt. Nach dem Ausgabefilenamen können beliebig viele Eingabefiles angegeben werden. Wenn das Programm ein Eingabefile nicht findet, wird es beendet nachdem das Root-File korrekt geschlossen wurde. Im Beispiel config-File wird das Wort „ende“ benutzt um das Programm zu beenden, nachdem alle Eingabefiles gelesen und bearbeitet wurden. (Ein File namens „ende“ gibt es nicht, also wird das Programm ordnungsgemäß beendet.)

Die Reihenfolge innerhalb des config-Files (erst Parameter, dann Detektorsummenkalibration, dann Fileliste) darf nicht verändert werden. Der Parser in LMF2Root ist noch sehr „einfach“ und verkraftet das sonst nicht.

LMF2Root ist für den User prinzipiell in zwei Teile unterteilt. Im ersten Schritt bearbeitet man Daten vom TDC. Wenn man eine LMF-Datei einliest (*Parameter 1 = 0*) wird der Code in *ProcessTDCData.cpp* abgearbeitet. Für jedes Event wird die Funktion *ProcessTDCData* angesprungen. In dieser Funktion wird normalerweise *detector\_stuff* aufgerufen. Dort werden entsprechend der Parametereinstellungen die TDC-Daten in Orte und MCP-Zeiten umgerechnet. Der User kann diese dann über die Struct *Ueber* (definiert in *Ueberstruct.h*) benutzen. Innerhalb dieser Struct gibt es 3 Structs jeweils eine für den Recoil-, den Elektronen- und den Projektildetektor. Die Detektor Structs beinhalten dann Arrays für x,y und die MCP-Zeit und noch einige andere Dinge (Definition in *Ueberstruct.h* anschauen). Man erreicht also z.B. den x-Wert des 2. Hits des Elektronendetektors durch:

```
Ueber->elec.x[1]
```

In *ProcessTDCData* hat man außerdem noch Zugriff auf das komplette TDC-array in der Form *tdc\_ns[channel][hit]*, die Anzahl der Hits ist im Array *cnt[channel]*. Es gibt den Eventcounter (*eventcounter*), der die absolute Eventnummer enthält und an die Parameterliste kommt man über *parameter[Nummer]* heran.

Der Plan in diesem Schritt ist normalerweise die Orte und Flugzeiten der Teilchen zu berechnen und nur noch den Teil der Daten wegzuschreiben, den man in der späteren Analyse benötigt. Die Daten werden hierbei wie die Hsistogramme ebenfalls in das Root-File geschrieben. Dies geschieht in der Zeile:

```
Hist->NTuple(9999, "Data", "H20BESSY08",  
"rlx:rlly:rltof:r2x:r2y:r2tof:elx:ely:eltof:e2x:e2y:e2tof", 32000,  
NTupleData);
```

*Hist->Ntuple* erzeugt und füllt ein Ntuple (so heißen bei Root die Gebilde, in denen Listmodedaten stehen). Die Zahl „9999“ ist intern zum Verwalten der Histogramme und Ntuple (man kann mehrere in eine Datei schreiben) benötigt und ist prinzipiell beliebig (sollte nur <10000 sein), muß aber für jedes Histogramm und Ntuple unterschiedlich sein. „Data“ und „H20BESSY08“ sind Name Titel des Ntuples. Danach kommen die Namen der Daten, die weggeschrieben werden sollen. Die Namen sind durch einen Doppelpunkt getrennt. Beim späteren wieder einlesen der Daten findet Root die Daten anhand dieser Namen wieder, sie sollten also sinnvoll sein. Im Beispiel hier schreibe ich also Orte und Flugzeiten von zwei Recoils und einem Elektron weg. Die

Daten selbst sind im Array `NtupleData` enthalten. Man muß immer darauf achten, daß die Größe des Arrays richtig ist, also in diesem Beispiel hier „9“. Das Array ist am Anfang von `ProcessTDCData` definiert. In das Array schreibt man dann die Daten, die in das Ntuple geschrieben werden sollen hinein, also z.B. `NtupleData[2] = rectof1`, nachdem man die Flugzeit des 1. Recoils ausgerechnet hat. Es gibt die bool `writeNTuple`, die am Anfang immer auf `false` gesetzt wird. Der Code sorgt im Moment dafür, daß Daten nur für `writeNTuple == true` weggeschrieben werden. Damit hat man also einen einfachen Presorter. Wenn einem die Daten gefallen, einfach `writeNTuple` auf `true` setzen, dann werden sie weggeschrieben.

Im zweiten Schritt liest man nun nicht mehr LMF-Dateien, sondern die Ntuples aus den Root-Files. Hierzu stellt man Parameter `1 = 1`. Jetzt wird **statt** `ProcessTDCData` die Funktion `analysis` in `Analysis.cpp` aufgerufen. Hier werden zu Beginn die Daten aus dem Ntuple geholt. Man muß sich für jede Variable, die man weggeschrieben hat ein `float` definieren und diese Variable dann mit der richtigen Spalte des Ntuples verknüpfen. Dies geschieht mit der Zeile:

```
Data->SetBranchAdress("r1x",&r1x);
```

Es wurde vorher ein `float` mit dem Namen `r1x` definiert. Dieses soll nun die Daten bekommen, die im Ntuple unter dem Namen „r1x“ abgelegt wurden. Der Name im Ntuple steht hierbei an erster Stelle mit den Anführungszeichen, die Verknüpfung mit der Variable ist das `&r1x` (wir übergeben einen Pointer auf diese Variable). Name und Variablenname müssen nicht gleich sein. Das Ganze wird dann noch mit den restlichen 8 Spalten des Beispiel Ntuples gemacht. Weiter unten im Code kann man dann ganz normal mit den Variablen rechnen oder sie in Histogramme einfüllen etc.

In diesem Teil der Analyse hat man tatsächlich nur die Daten, die man vorher in das Ntuple geschrieben hat. Das TDC-Array hat man also nicht mehr. Man muß also wirklich alles, was man später braucht in `ProcessTDCData` in das Ntuple schreiben!

## Histogramme

Es gibt 1d, 2d und 3d Histogramme in `LMF2Root`.

### 1d-Histogramm:

```
Hist->fill1(id, name, fillX, weight, title, nXbins, xLower, xUpper, titleX, dir);
```

Hierbei ist `id` wieder eine Zahl < 10000, die intern benötigt wird und für den User prinzipiell keine Bedeutung hat. Sie darf aber, wie gesagt, für jedes Histogramm nur einmal vorkommen, da `LMF2Root`-intern auf das Histogramm über diese Zahl zugegriffen wird. Am besten nummeriert man seine Histogramme einfach grob durch. Wenn man aus Versehen zwei verschiedenen Histogrammen die gleiche Zahl gibt, spuckt das Programm beim Ausführen eine Warnung aus. Mit `name` gibt man dem Histogramm den Namen, unter dem man es dann in Root findet. Reine Zahlen gehen als Namen nicht, das erste Zeichen muß immer ein Buchstabe sein (also z.B. „h100“ geht). Die Variable, die geplottet werden soll, ist `fillX`, mit `weight` gibt man an, um wieviel das Bin hochgezählt werden soll, also im Normalfall 1. Der Titel des Histogramms (erscheint beim Plotten oben links) ist `title`, wenn man keinen Titel haben will muß man "" angeben. Dann kommt die Histogrammdefinition mit der Anzahl der Bins und der unteren und oberen Grenze (`nXbins`, `xLower` und `xUpper`). Mit `titleX` kann man die x-Achse beschriften. Man kann Histogramme in Verzeichnissen innerhalb des Rootfiles ablegen. Wenn man das tun will, muß man mit `dir` das Verzeichnis angeben. Man kann `dir` auch komplett weglassen.

### 2d-Histogramm:

```
Hist->fill2(id, name, fillX, fillY, weight, title, nXbins, xLower, xUpper, titleX, nYbins, yLower, yUpper, titleY, dir);
```

Analog zum 1-d Histogramm, nur das es nun noch eine y-Achse mit all ihren Parametern gibt.

Um z.B. ein Ortsbild zu erhalten macht man:

```
Hist->fill2(15,"Det", pox, posy, 1., "Ortsbild", 200, -50., 50., "x[mm]",  
200, -50., 50., "y[mm]");
```

Dieses Histogramm findet man dann unter dem Namen „Det“ im Rootfile und es werden die Orte *pox* und *posy* eingefüllt.

### 3d-Histogramm:

Es gibt in Root 3d-Histogramme, also ein Histogramm, in dem dargestellt wird, wie oft die Kombination aus drei Variablen vorkommt:

```
Hist->fill3(id, name, fillX, fillY, fillZ, weight, title, nXbins, xLower,  
xUpper, titleX, nYbins, yLower, yUpper, titleY, nZbins, zLower, zUpper,  
titleZ, dir);
```

Bei der Histogrammkategorie handelt es sich um eine abgewandelte Version der Histogrammkategorie von Lutz. Ich habe die Funktionen (noch mehr) überladen. Man kann daher auch den Syntax von Lutz benutzen, was vielleicht Leuten, die sich schon an eine Acquiris-Auswertung gewöhnt haben hilft.

Man kann statt „fill1“, „fill2“, „fill3“ auch einfach nur „fill“ benutzen, der Compiler erkennt dann an der Anzahl der übergebenen Variablen, welche Funktion gemeint ist.

## Detektor kalibrieren und Resort

Die neue Resortroutine macht sehr viele Dinge automatisch. Die für die Kalibration benötigten Spektren werden ebenfalls alle automatisch erzeugt. Hier eine Liste der Dinge, die man tun muß, um einen Detektor zu kalibrieren und die Resortroutine richtig zu benutzen:

1) Für alle Detektoren: Sort Routine aus. Auto-Kalibration aus.

2a) Skalenfaktoren alle auf den gleichen Wert stellen (z.B. 1).

2b) *w\_Offset* auf 0 stellen.

2c) Zeitsummenoffsets auf 0 stellen.

2d) *runtime* auf 200 stellen.

2e) Ortsnullpunktverschiebung auf 0 stellen (x und y).

3) Rohdaten plotten und dann:

3a) die Zeitsummenpositionen und halbe Fussbreiten finden und im config-File eintragen.

3b) jeweils alle 3 Skalenfaktoren mit einem gemeinsamen Faktor multiplizieren, so dass das Ortsbild einen Durchmesser hat, der dem MCP-Durchmesser entspricht.

4) Daten nochmal plotten.

4a) Zeitsummen kontrollieren (müssen jetzt bei Null liegen).

4b) Ortsbild checken. Stimmt der Durchmesser?

4c) Im Ortsbild den Nullpunkt finden. Gemeint ist der Mittelpunkt des MCP und NICHT der Mittelpunkt der Physik. Das ist wichtig.

4) Nochmal Daten reinlaufen lassen. Ortsbildverschiebung checken.

4a) Jetzt *u1-u2* (in Nanosekunden) und *v1-v2* und *w1-w2* anschauen. Am besten y-Achse logarithmisch darstellen.

4b) Jetzt muss ein Wert fuer "Runtime" gefunden werden. Man muss in den Bildern schauen, wo der Detektor aufhoert (links und rechts). Was zaehlt, sind die Absolutwerte. "Runtime" ist dann der groesste dieser 6 Werte. Diesen Wert dann im config-File eintragen.

4c) Die folgenden Punkte für jeden Detektor einzeln durchführen, also immer nur ein Detektor gleichzeitig: Auto-Kalibration anschalten und Daten reinlaufen lassen. Dann bekommt man neue Werte fuer dei

Skalenfaktoren vom Layer *v* und *w* genannt und den *w\_offset*. Diese Werte im config-File eintragen und 4c wiederholen noch einmal wiederholen. Wenn man es ganu haben will, dann macht man es noch ein drittes mal.

4d) Bei der letzten Auto-Kalibration wurde auch ein neues Text-file geschrieben, das die Korrekturpunkte fuer

die ortsabhaengige Zeitsummenkorrektur enthaelt. Dieser Teil muss mit copy/paste in das config.txt eingetragen werden. Bitte auf die Detektorreihenfolge achten! Sonst werden evtl. die falschen Korrekturdaten auf den falschen Detektor angewendet.

5) Auto-Kalibration ausschalten und Sortieren anschalten. Beten. Fertig.

Für jeden Detektor werden die folgenden Histogramme erzeugt, der Typ des Detektors steht immer am Anfang des Namens (also z.B. „rec“ für den Recoildetektor):

**\_mcp\_hit:** Anzahl der Hits auf dem MCP-Kanal vor der Resortroutine.

**\_u1\_hit:** Anzahl der Hits auf dem Kanal des einen Endes des u-Layers vor der Resortroutine.

**\_u2\_hit:** Anzahl der Hits auf dem Kanal des anderen Endes des u-Layers vor der Resortroutine.

**\_v1\_hit:** Anzahl der Hits auf dem Kanal des einen Endes des v-Layers vor der Resortroutine.

**\_v2\_hit:** Anzahl der Hits auf dem Kanal des anderen Endes des v-Layers vor der Resortroutine.

**\_w1\_hit:** Anzahl der Hits auf dem Kanal des einen Endes des w-Layers vor der Resortroutine. (Wird nur erzeugt, wenn der Detektor eine Hexanode ist.)

**\_w2\_hit:** Anzahl der Hits auf dem Kanal des anderen Endes des w-Layers vor der Resortroutine. (Wird nur erzeugt, wenn der Detektor eine Hexanode ist.)

**\_sumu:** 1-dim Zeitsumme des u-Layers.

**\_sumv:** 1-dim Zeitsumme des v-Layers.

**\_sumw:** 1-dim Zeitsumme des w-Layers. (Wird nur erzeugt, wenn der Detektor eine Hexanode ist.)

**\_u:** 1-dim Ort auf dem u-Layer in ns.

**\_v:** 1-dim Ort auf dem v-Layer in ns.

**\_w:** 1-dim Ort auf dem w-Layer in ns.

**\_sumu\_u:** Ort auf dem u-Layer (in ns) gegen Zeitsumme des Layers.

**\_sumu\_v:** Ort auf dem v-Layer (in ns) gegen Zeitsumme des Layers.

**\_sumu\_w:** Ort auf dem w-Layer (in ns) gegen Zeitsumme des Layers. (Wird nur erzeugt, wenn der Detektor eine Hexanode ist.)

**\_uv\_mm:** Ortsbild aus u und v Layer berechnet in mm.

**\_uw\_mm:** Ortsbild aus u und w Layer berechnet in mm. (Wird nur erzeugt, wenn der Detektor eine Hexanode ist.)

**\_vw\_mm:** Ortsbild aus v und w Layer berechnet in mm. (Wird nur erzeugt, wenn der Detektor eine Hexanode ist.)